DPM: A MEASUREMENT SYSTEM FOR DISTRIBUTED PROGRAMS

by

Barton P. Miller

Computer Sciences Technical Report #592

April 1985

DPM: A Measurement System for Distributed Programs

Barton P. Miller

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

DPM is a system for monitoring the execution and performance of distributed programs. An important characteristic of its design is the simplicity of each part of the design. This simplicity has resulted in a system of tools that has wide range of applications and that was relatively easy to construct.

First, we start with a simple model of distributed computation based on message interactions. We use this model of computation to develop a structure for a measurement tool for distributed programs. The tool structure was used to guide the implementation of DPM for both the DEMOS/MP and Berkeley UNIX operating systems.

The power of DPM is shown in two directions. The first is the type of performance information that can be obtained by using DPM. This information currently includes measuring communication statistics, dynamic program structure and parallelism. The second direction is the flexibility of the tool. DPM can be used for post mortem analysis of a program's performance, real time performance monitoring, and generating data to be used by the operating system for such things as a scheduler for load balancing.

Research supported by the National Science Foundation grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235

So I quoted the first law of the Mentat at her: "A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it."

Paul Mau'dib in Dune
by FRANK HERBERT

1. Introduction

This paper presents a framework for a system to measure the performance of distributed programs. This framework includes a model of distributed computation, a description of the measurement principles and methods, and a guideline for implementing these ideas. We have constructed a measurement system (called the Distributed Programs Monitor, or DPM) based on these concepts. DPM has been implemented and used for measurement studies on two different operatings systems (DEMOS/MP [Miller, Presotto & Powell 85] and Berkeley UNIX[Joy *et al* 83]).

Collecting data about a program's performance is not enough; we must supply some form of interpretation or analysis of the data. We include, as part of DPM, several analysis techniques that can provide information about the structure, the amount of parallelism, and the communications patterns of a distributed program.

DPM is more than a particular implementation of a measurement facility. It provides a framework for other activities that are based on the monitoring of a distributed program. Some of these activities include real time monitoring and display of the activities of a program, and use of the measurement data for feedback scheduling activities such as load balancing.

1.1. Overview

The driving principle in the design of DPM is simplicity. The model of distributed computation is simple in the sense that it is general enough to make it applicable to a wide range of systems. Our methods of measurement are simple to insure easy implementation. The implementation of our tools is simply structured to provide confidence in their correctness.

The goal of simplicity has produced two subordinate goals, *consistency* and *transparency*. The goal of consistency requires us to maintain a constant view as we progress from model of computation, to measurement system, to implementation, and finally to analysis. For example, suppose with wish to monitor a program that was sending and receiving messages. The programmer would use SEND and RECEIVE primitives. The measurement system should be based on these primitives, and the analysis procedures should work at the same level. We do not want the measurements to be based at a lower level, such as the network protocol, as this could overwhelm the programmer with information about packets, message routing, and bit error detection. Likewise, we would not want to base the measurement on a high level of semantics, such as formal descriptions of program behavior [Baiardi *et al* 83]. This might delete valuable information. Our model does not preclude the use of formal descriptions; we could still use a formal description of program behavior but this description would be based on the primitive events (in this example) of message SEND and RECEIVE.

The goal of transparency enforces simplicity of use for the programmer. To measure a program we should not have to recompile, re-link, or write in a special style or language. We should not have to supply special information to the measurement system to have it function correctly[†]. Transparency also means that the performance of the program being monitored is not significantly disturbed. A monitor built in software will always have some affect on a program's performance, but our design goal is to minimize this effect. This goal will influence both the design and the implementation of the measurement system.

Our measurements are done passively, as opposed to systems that interact with the computations — such as is the case with interactive debuggers. By this, we mean that actions such as redirection of messages, breakpoints, and modifications of the message streams are not allowed. DPM is an observer of the computation, and not a participant. There are designs (for example, see [Schiffenbauer 81]) that provide transparent control and monitoring of interprocess communication. The complexity of such a design and the difficulty of implementing on a multi-user system has

† Note that we say "have to". The option is still available to augment the measurement system with, e.g., compiler supplied information

made this alternative unattractive.

1.2. What is a Distributed Program? And Other Definitions

Our model of distributed computation provides the guidelines for the design of DPM. It is not a formal model in that we do not use it as the basis for mathematical analysis; rather, the model can be considered as a reference point for the design and implementation of the measurement system.

We define a *distributed program* to be a collection of processes cooperating to perform some computation. The component processes are not constrained to run on the same machine. No assumptions are made about the locations of the processes. The two extremes are the case where all processes run on the same machine, and the case where each process runs on its own machine. The tools and methods that we are describing do not depend on how the program is distributed.

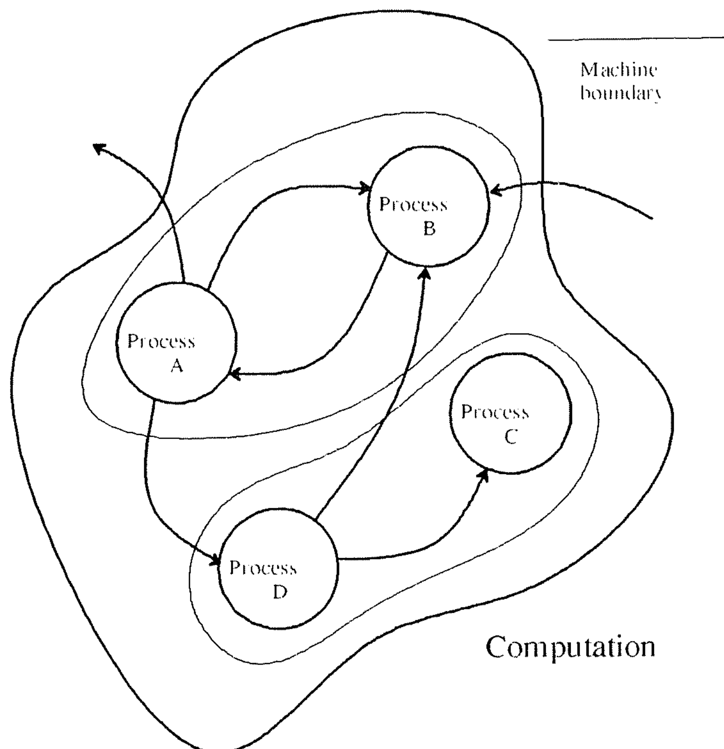

Figure 1.1: A Distributed Computation

A distributed program (more simply called a *computation*) is made up of *processes* which are the basic building blocks of a computation. A process consists of an address space containing code and data, and an execution stream. Each process has access only to its own address space. Processes do two things: compute and communicate. Computing is the normal execution of instructions and does not affect the state of other processes. These instructions are referred to as *internal events*. Communication is the means by which a process will interact with other processes and the operating system. Interactions are referred to as *external events*. The complexities of the distributed environment become apparent when a process in a computation interacts with another part of the computation. A computation is illustrated in Figure 1.1.

Communication is based on messages. A message allows the copying of part of one process's address space into that of another process. A message is an interaction involving exactly two processes: the process originating the data (the *sender*) and the process consuming the data (the *receiver*). We make no restrictions on the structure of the message delivery. The communications path may be unidirectional or bidirectional. The message passing operations may be synchronous or asynchronous. Message delivery may or may not be guaranteed or required to preserve message order. Message paths may be dynamically or statically created and destroyed, and the processes in the computation may be dynamically created and destroyed. We make no assumptions about the network or facility underlying the communications mechanism. Our model of computation applies to a wide range of systems because of its simplicity.

Our model of computation does not include systems that have processes with shared address spaces. Conceptually, a shared memory system can be modeled as a message based system (and vice versa)[Lauer & Needham 79]. In practice, the interactions in a message based system are generally easier to detect than in a shared memory system, and therefore easier to monitor.

Processes execute on *machines* that do not have direct access to each other's memories. Each machine has a portion of the operating system running on it to support process execution, communications, memory management, and device management. The communication functions supplied by

the operating system provide for interprocess communications both within and between machines.

2. The Measurement Structure

Our structure of measurement follows the basic philosophy of "look, but don't touch" with respect to the program that is being studied. The goal is minimal disturbance of the execution of the program. This means that the computation being measured should not execute more slowly or achieve different results because it is being measured. If the cost of measurement is high, then the act of measuring a computation could substantially change its execution behavior.

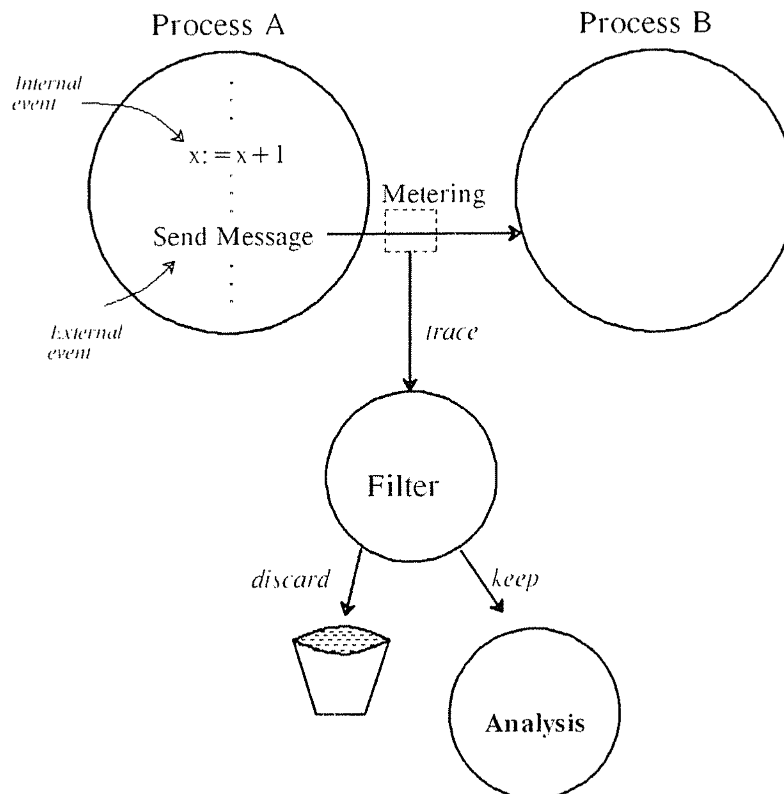

Figure 2.1: Events and the Measurement Model

Figure 2.1 gives an overview of event detection and our measurement model. Internal events are not visible from outside of a process and are therefore not detected. In our measurement structure, the detection of external events is referred to as *metering*. A trace is produced for each event

that is detected. After the trace is produced, a decision is made whether or not to keep the trace. The selection decision is called *filtering*. If the trace is kept, it is stored until it processed to provide results that may be used to understand the behavior of the process (and the overall computation). We call the processing of the traces *analysis*.

The metering stage of measurement will lie within the kernel of the operating system because of the desire not to change the program itself. The facility should be simple, so as to make the necessary modifications as simple as possible. Changes to an operating system kernel are typically much more difficult than those to parts of the system outside the kernel†. Alternatively, the event detection could be placed in the language runtime library, compiler generate code, or inserted directly by the programmer. While these methods may be simpler to build, they provide for less generality and less transparency. For example, these alternatives might require the programmer to use a particular language or might allocate one of the available file descriptors.

The filtering stage provides for a flexible set of rules to perform data reduction. This facility allows easy change to the selection criteria and easy adapting to new or changed trace types.

The analysis of the data provides a summary of execution of the computation. It is at the analysis stage that useful information is provided about the computation. The goal of the measurements dictate the type of analysis being performed and the overall structure of the measurement system (see Section 4).

Our design is similar in structure to the METRIC system[McDaniel 75]. The separation of function used in METRIC provides flexibility in the location of data detection, selection, and analysis. Our design differs from METRIC in several ways. First, METRIC was not transparent; programmers had to explicitly insert trace calls into their programs. Second, METRIC used a different model of distributed processing. The design of METRIC incorporates the concept of a broadcast network media, with the structure of the network visible to the measurement tools.

---

† As a rule of thumb, the effort necessary (including design, coding, and testing) to put a given function in the operating system kernel is 10 times greater than implementing the same function outside the kernel (in a process). A similar statement can be made when moving kernel functions into microcode

3. The Measurement Facility

The measurement facility is described by the events that we measure and the structure of the measurement tools. The measurement tools consist of the previously mentioned components (meter, filter, analysis) and a user interface. We describe the events, meter, filter, and user interface in this section. The analyses are described in Section 4.

3.1. Events and Trace Records

There is a set of meter events that reflect the basic operations as seen by the programmer of a distributed computation. The structure of the metering stage is very simple due to the small set of meter events (currently about 10). These event types are, for the most part, the same across the different operating systems supporting the measurement facility. These events consist primarily of activities that reflect interactions between processes (such as a message being sent and received). Other events related to communications are also recorded. This group of events consists of actions that effect the creation, modification, and destruction of communications paths. The last group of events that are recorded pertain to the state of the processes in the computation. The basic events are the creation of a process, the starting and stopping of its execution, and the destruction (termination) of the process. Depending on the system from which the measurements are being extracted, there may be slight variations in the details of the data collected with each event type.

METRIC (and other systems) allowed the users to specify their own event trace types. It would be easy to add such a mechanism to DPM. Only the meter would need to be changed and these changes would be minor. We chose to not provide this facility for reasons of transparency. User defined traces would require explicit use of the trace facility within the processes being measured.

Included with each event trace is a standard header describing the trace. The header of an event trace contains the following fields:

MACHINEID The machine from which the trace came.

PROCTIME	The amount of CPU time used by this process up to the time this trace was generated. PROCTIME is independent of the load on the host system.
TRACETYPE	The type of event described by this trace.
PC	The program counter indicating the location in the process causing the event.

The event trace types are: SEND, RECEIVECALL, RECEIVE, MESSAGEQUEUED, CREATEPATH, DESTROYPATH, CREATEPROCESS, STARTPROCESS, STOPPROCESS, and DESTROYPROCESS.

The meter traces do not include the contents of the messages sent by the users. The meter traces record only the occurrence of an event and information about which processes were involved. This results in less trace data to communicate and store.

Even though we do not include the message contents in the traces, we can still detect activity within the processes that are being measured. By using the PC information in the trace header we can identify the specific procedure within the process that caused the event.

3.2. Metering

Metering is the activity that takes place within the operating system kernel to extract the events of interest. This portion of the measurement facility is the only change required to the supporting operating system. There are two basic parts to metering. The first part is the collection of the information that is needed to form a trace. The second part is a communications path over which to pass the trace information.

The point in the operating system kernel where the necessary information is available must be located to meter the specified events. At these points, we insert *meter probes* in the code of the kernel. These probes are procedure calls to a software module (*meter module*) that is responsible for passing the traces to the filter. The parameters particular to the specific trace being generated are passed to the meter module with the procedure call. These values, along with the standard format header, are passed to the filter.

A communications path, called the *meter path*, must be available to the metering routines for sending traces to the filter stage. This meter path should be reliable in the sense that messages sent are eventually delivered. Messages should not be lost or duplicated. It is possible that later stages of DPM (the analysis stage) might be able to detect such anomalies in the data, but it is a good policy to try to reduce these problems as much as possible. There is no constraint on message ordering. Each trace includes the local machine time; hence the analysis stage is able to restore (partial) message order.

There must be a method for allowing the programmer to specify the message path to the filter stage for processes being metered. This requires the addition of a new system function (system call) allowing a message channel to be specified.

Metering, to the extent possible, should not increase the complexity of the host kernel. In fact, few procedure calls are added in the existing host kernel code. The number of these calls inserted in the kernel is about the same as the number of different types of traces. In the same spirit of minimizing complexity, the metering stage uses the same message facility as already exists in the host operating system for its communication channel to the filter stage.

We try to minimize the performance overhead of generating the trace messages. Two mechanisms help in this effort. The first is buffering of the trace messages. The major cost in generating the traces is sending the message over the meter connection. We typically buffer up to 50 traces (for each process) before sending a trace message. The second mechanism that contributes to the performance is that the meter module can access the communications routines with substantially less overhead than could a process.

We also try to minimizing the cost of *not* generating traces. This is the cost incurred for a process that is not being monitored. One extra test and conditional branch (an "if" statement) is added to each system call activity. This is not enough to cause a measurable change in performance.

3.3. Filtering

Filtering is the data selection and reduction stage in the measurement system. It reduces both the size and number of traces as they are produced. Data is received from the metering stage, filtered, and then passed on to the analysis stage or stored for later analysis. The scheme currently used in the measurement system is based on a general, one level, pattern matching algorithm. More sophisticated schemes (such as presented in [Bates & Wileden 83]) may be used in subsequent versions of DPM. This filter also allows for the specification of trace record formats, so that the filter can process traces coming from different systems.

The location of the metered processes and the location and number of filter processes are a source of flexibility in the measurement system. The choice of how to attach processes to filters provides the ability to do many types of analyses (see Section 3.3.3).

3.3.1. Trace Description and Selection

The filter receives three types of input. These are the *descriptions* of the trace record formats, the *selection rules* for filtering, and the trace records themselves. The descriptions and the selection rules are processed at the time the filter begins its execution. These stay in effect while the trace records are processed.

```
header 0,
     machine,0,4,16
     traceType,4,4,10
     procTime,8,4,10
     pc,12,4,10;
SEND 1,
     fromTask,0,4,16
     toTask,4,4,16
     channel,8,4,10
     code,12,4,10
     sendLink,16,4,10
     passLink,20,4,10;
```

Figure 3.1: Trace Description

A trace record consists of a header, standard for all types of traces, and an type dependent portion. The trace descriptions consist of a description of the header format, and a description of the type dependent part of each trace. Figure 3.1 gives a sample of a description for the header and a SEND trace (for the DEMOS/MP version). The first part of the trace description gives the name (in this example ``header'' or ``SEND''), and the value identifying that trace type. Following this is an entry for each field in the trace. Each entry contains the name of the field (to correspond with those that will be specified in the selection rules), its offset into the message, the length of the field, and a default number base for displaying the field.

Using the descriptions, it is possible to take traces from different systems, in different formats, and process them in a uniform manner. As the filter receives each trace, it compares the trace against a set of selection rules. Each rule is a pattern that, if matched, specifies that the current trace is to be accepted. A rule is a list of selection fields which specify the conditions for acceptance of each field. A selection field is a field name, a selection condition (comparison operator), and a value. A selection field is satisfied if the evaluated condition is true. If all selection fields in a selection rule are true, the trace is accepted. The possible conditions are $>$, $<$, $=$, $\neq$, $\geq$, and $\leq$. Figure 3.2 shows a simple set of selection rules.

```
machine=3, traceType=1, time>10000;
machine=1, type=1, fromTask=20003, toTask=30003 channel=0;
```

Figure 3.2: Simple Selection Rules

The value specified with each selection field has several options. The value may be either a simple value, a *wildcard* ``*'' (a value that matches any value), or a field name. In the case where the value is a field name, the value of the field specified for the selection field is compared to the named field (also contained within this record). Any value may be prefixed with the the *discard* indicator ``#'', so that if the trace is accepted, this field is eliminated from the trace. This allows the size of the traces to be reduced. Figure 3.3 shows a more interesting set of selection rules.

```
machine=#*, traceType=1, time=#*, procTime=#*, code=#*;
        type=8, readTask=writeTask, size≥1024;
```

Figure 3.3: Selection Rules

3.3.2. Early Filtering

A crude level of filtering is done before the trace messages leave the kernel (metering stage). The motivation for this is to reduce the volume of trace messages if the tracing paradigm permits. The metering causes no significant performance decrease for the process being metered or for the host kernel; the early filtering is a simple selection.

For each operation that can be traced (e.g., SEND), there is a flag which indicates whether that operation is to be metered. If this flag is not set, no traces of this type are generated for the filter. If an analysis required only the events of a message being sent and received, then all other trace types could be ignored, reducing the effects of the measurements on system performance.

The selection flags are combined into a bit mask and stored with the description of the meter path for each process. There must be a system function (system call) that allows the selection flags to be specified.

3.3.3. Configurations

Figure 3.4 shows a sample configuration for metering user processes. Several processes forming a computation are sending traces to a single filter, which is selecting and storing the data for later analysis. We would expect this structure to be useful to a programmer evaluating a new program.

Different configurations provide for the ability to apply the measurement system to different problems. For example, the configuration in Figure 3.4 could be extended to have the filter collect data on all communications activities within a single machine. This type of configuration could allow measurement of message quantity and frequency, queue lengths, and process scheduling. Our measurement system can gather these different types of information which traditionally required specialized tools to be built. It takes no extra work to extend this type of measurement to a collection of machines, or to the entire system.

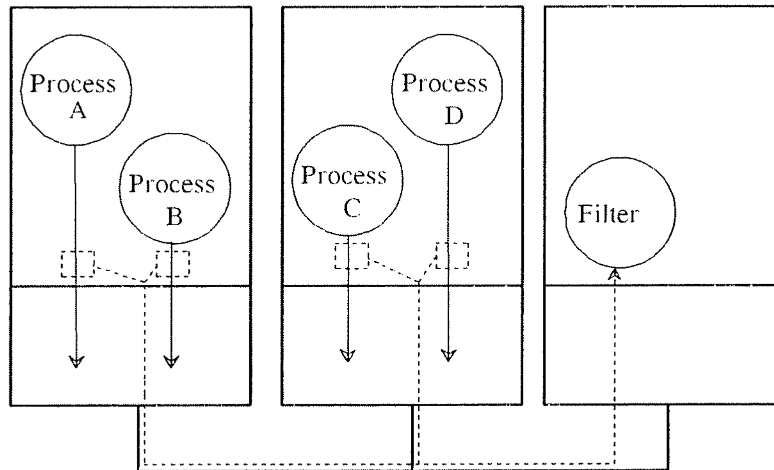

Figure 3.4: Computation with Single Filter

If network (communication) load is critical, then we can have a filter on each machine and merge the trace records when the computation has terminated. If CPU load is critical, then the filter process(es) can be placed on its own machine.

An alternative to the post mortem processing of the traces is to process them as they are produced and selected by the filter (in *real time*). This is discussed further in Section 5.

3.4. User Interface

The user interface to DPM is a command interpreter that allows the programmer to specify the (1) program (processes) to run, (2) events to monitor, (3) name of the filter (with descriptions and templates, if the standard filter is used), and (4) the analyses to be run on the trace data after it is collected. A complete description of the command language and structure of the user interface is given in [Macrander 84, Miller *et al* 85].

4. Analysis Techniques

A collection of data needs some form of interpretation to have some meaning. A basic tenet of this paper is that the measurement model and techniques, and the associated tools, can provide useful data. To demonstrate this, we describe several approaches for the analysis of the trace data generated by our measurement system.

4.1. Basic Communications Statistics

We have defined a computation to be a collection of cooperating processes. The processes cooperate, and the cooperation is based on some communications mechanism. It is reasonable then to want to know the nature of the communications between processes. Several basic questions come to mind. Who is talking to whom? (Which processes are talking to which other processes?) What is the volume (total message traffic) of the communications? How frequent (time density) are the communications? How large are messages?

In addition to these basic questions, a few more interesting queries come to mind. Given information about the arrival and consumption of messages, we can derive information about the message queues. It is possible to gather statistics such as the maximum, and average queue lengths for each process. With the same information we can obtain the minimum, maximum, and average time that a message waits in the incoming message queue before it is consumed by the process. However, we are not restricted to minimum, maximum and average. We can collect data to record the distribution of the various measurements.

There are several reasons why these message statistics are useful. When first studying a distributed program (or the traces of a distributed program's execution), it is useful to get an overview of its behavior. The message quantity and density statistics give a first view of the interactions in the computation. This gives the programmer an initial indication of the behavior of the program. Information such as local (intramachine) and remote (intermachine) message levels can also be obtained.

The message queue length information can help in structuring the program. Suppose we have a program that provides some service by receiving request messages, performing some action, and then replying to the requesting process. If many requests are being handled at one time, then the server program may involve several processes working together. If we know that the message queues for awaiting requests are large, then we should structure the server to handle requests differently, or allocate more resources (e.g., machines) to the server.

4.2. Detecting Paths of Causality

When we write a computation consisting of several processes, we specify the order and frequency of the communications in the computation. We specify this information for each interaction between processes. We establish rules and protocols to provide for the correct execution of the program. But when the entire computation is executing, the overall interactions are more complex than this static picture of the computation suggests. The increased complexity comes from parallel execution within the computation and from the fact that several partially completed activities maybe be simultaneously active within the computation.

The model of computation in which we are interested for this analysis is that of a *server*. A server is a computation that receives requests from processes outside the computation, computes a result (involving one or more of the processes within the computation), and then returns the result to the requesting process. There are several questions to be answered about the behavior of a server. One questions is: given a request message received by the server, what message paths within the server are most commonly traveled? This question can be translated to: what sequences of interprocess communications occur most frequently? For sequences of length two, we can derive this information from the basic message statistics (see Section 4.1). The message statistics cannot provide information about longer sequences of interactions. Related to the longer sequences of interactions is a second question: given a process that has just received a message, where will that process next send a message? This question can also be viewed as determining a branching probability, given a specific input to a process.

The basic strategy for causality analysis is to identify each request to the server, and follow the sequence of interactions within the server caused by that request. To do this, we first collect SEND and RECEIVE traces and a graph of the events in the processes is constructed, with arcs in the graph between each corresponding send and receive (see Figure 4.1). This graph represents the complete collection of interactions between processes during the life of the computation.

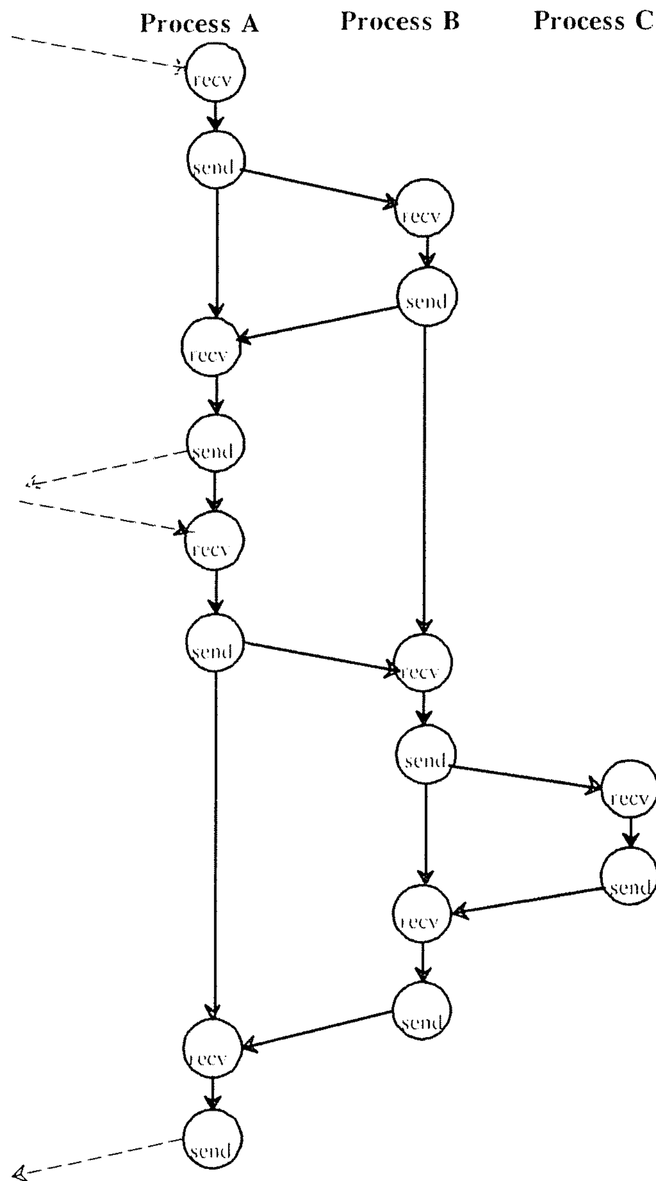

Figure 4.1: Sample Computation Graph for Causality Analysis

To reduce the complexity of analyzing such a large quantity of data, we convert the problem to one of manipulating character strings. Each process in the computation is assigned a single letter

designator. The two events corresponding to a message being sent and received are designated by the letter for the sending process, followed by the letter for the receiving process. For example, the first send and receive pair in Figure 4.1 would be the string:

AB

We create a list of strings, where each string represents one request to the server and the subsequent activity within the server. These strings are called *causality strings*.

There are three types of processes that are visible during a causality analysis. The first type of process is the *server* process. This process is contained within the computation that is providing service. The second type of process is the *requestor* process. Requestor processes are the customers of the server. They make requests and receive results. The last type of process is the *system* process. System processes are those processes to which the server makes requests. The system processes may be other servers, or perhaps a host kernel. Messages received from a requestor indicate a request for service from the server computation. Messages to system processes from the server are ignored (as are the responses), as we are interested in tracing the flow of control through the server, and not through external processes.

The algorithm for building the causality strings traverses the entire computation graph. The list of causality strings is built by:

(1) searching for each message receive event from a requestor process;

(2) for each such receive, the message sends immediately following the receive are identified;

(3) the message receives corresponding to sends in (2) are identified (i.e., the message arcs are followed), and steps (2) and (3) are repeated for each receive.

A causality string is initially a single character, which is the process performing a message receive that was detected in step (1). Each time a send is followed (i.e., a message arc is traversed) to its corresponding receive (step (2)), an additional letter (identifying the receiving process) is added to the causality string. Events associated with system processes are ignored in this algorithm. For example, the causality strings for Figure 4.1 are:

ABA
ABCBA

Once we have the causality strings, there are several results that we can obtain from them. The first result is identifying the most commonly traveled paths through the server. We store these strings in lexicographical order with a value indicating the number of times that the string has occurred in the computation. This list of strings identifies the most commonly occurring message sequences.

Given three processes, A, B, and C, we use the causality strings to compute the probability that process A, having just received a message from process B, will next send a message to process C. This information is obtained by generating all of the substrings of length three, and then tabulating them. The probabilities cannot be calculated from the simple message statistics since these statistics do not correlate message receives with the corresponding message sends.

This analysis technique was used in a study of the DEMOS/MP file system[Powell 77, Miller, Presotto & Powell 85]. The DEMOS/MP file system consists of four processes (Request Interpreter, Directory Manager, Buffer Manager, and Disk Interface) which function together to provide a file service to user processes.

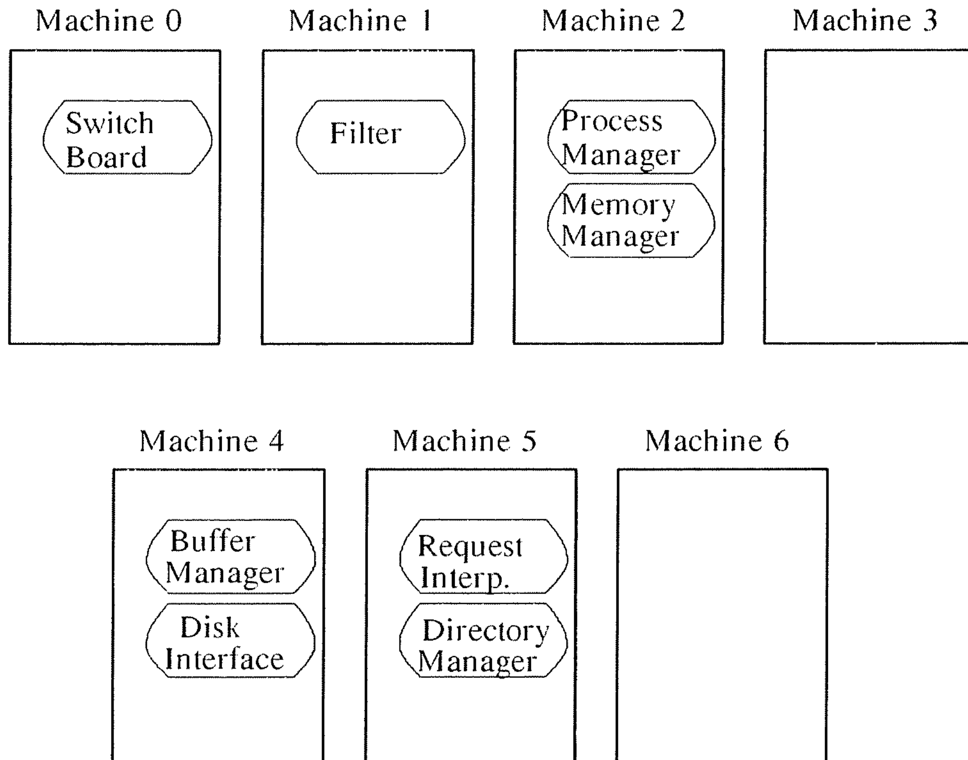

Figure 4.2: Layout of System Processes for File System Test

The file system was run under heavy loads (many user processes) and its execution was monitored by DPM. The configuration for the measurement is shown in Figure 4.2. User processes were distributed among most of the machines. The trace data that was collected was used to build the graphs and strings described previously. Substrings of length three were used to compute the probability of messages flowing between the file system processes and these probabilities were used to construct the diagram in Figure 4.3. This figure shows the flow of data and control within the file system.

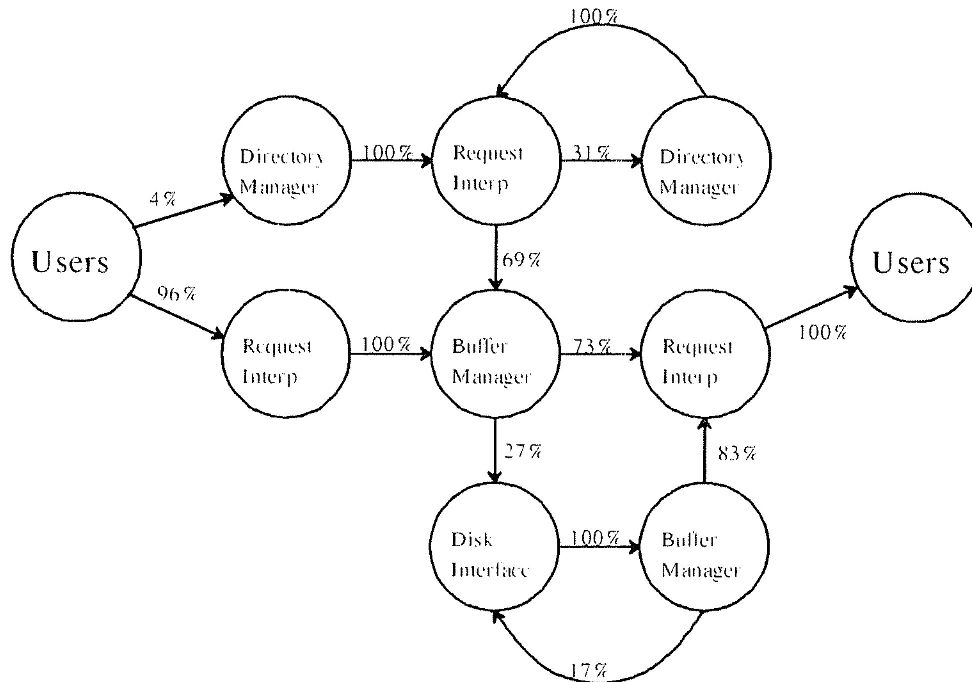

Figure 4.3: File System State Diagram

The interesting result is that by using a general performance tool, such as DPM, and knowing nothing about the internal structure of the file system, we can obtain valuable information about its internal operations.

For example, from Figure 4.3, we can conclude:

(1) Messages from users go to the Directory Manager only 4% of the time (corresponding to file opens). This provides us with the ratio of file opens to reads/writes.

(2) The Request Interpreter asks the buffer manager for data, and 73% of the time a result is immediately returned. 27% of the time the buffer manager must ask the Disk Interface for the data. This represents the buffer hit rate (cache efficiency).

(3) The Buffer Manager will ask the Disk Interface for additional blocks of data 17% of the time, representing the frequency of following indirect references on the disk.

The above results give important structural information when provided to the programmer/analyst of the file system. These results were provided without the need for specialized measurement tools.

4.3. Measuring Parallelism in a Computation

One motivation for writing a distributing computation is to achieve an increase in its speed of execution. This performance increase is obtained by means of parallel execution of the processes within the computation. Once we construct a distributed program, the problem becomes: how do we measure the amount of parallelism in the execution of our program? In addition, it is useful to be able to project the amount of parallelism that can be achieved as we vary the location of processes among machines.

The algorithm for the analysis of parallelism uses traces obtained from a particular execution or executions of a computation. We can obtain these results even on systems that are currently running other users. The techniques used for this analysis are described in detail in [Miller 85] and a study using this analysis is described in [Lai & Miller 84].

5. Variations on a Theme

5.1. Realtime Monitoring

In Section 4 we assumed that the data analyses were performed after the computation had completed. The structure of DPM provides the flexibility to use analysis routines that process trace data as it is generated.

Consider the analysis of communication statistics presented in Section 4.1. This analysis could be easily adapted to execute in real time. Associated with the analysis routine would be a graphic display of the activities within the user program. Communication traffic are represented by colored arcs between process nodes. As traffic levels increased, the color of an arc progresses from violet to red, thus identifying active or *hot spots* in the program. Other communication statistics, such as average message sizes or input queue lengths, could also be displayed in this manner. The processes are also colored — representing the amount (percentage) of time that the processes execute.

5.2. Feedback Scheduling (Load balancing and other uses)

The analyses discussed thus far are intended to provide the programmer or system manager with information. This information may result in the programmer restructuring the computation being studied or the manager changing the environment in which the computation executes. In both cases, a human being is part of the feedback loop.

It is possible to return the results of some of the data analyses directly to the host system. After the meter traces have been analyzed and reduced to some reasonable statistic, this information can be passed directly to the host system to be used in scheduling decisions. The trace data becomes direct feedback to the host operating system.

We can use this structure to collect communications load data for process load balancing and file migration decisions. The data collection for feedback scheduling looks similar to that described in the previous section on system communications measurements. The same type of organization for metering and filtering could be used. The filters would not store the selected data. After filtering, the selected data would be passed on to the host system to provide data for scheduling. This is illustrated in Figure 5.1.

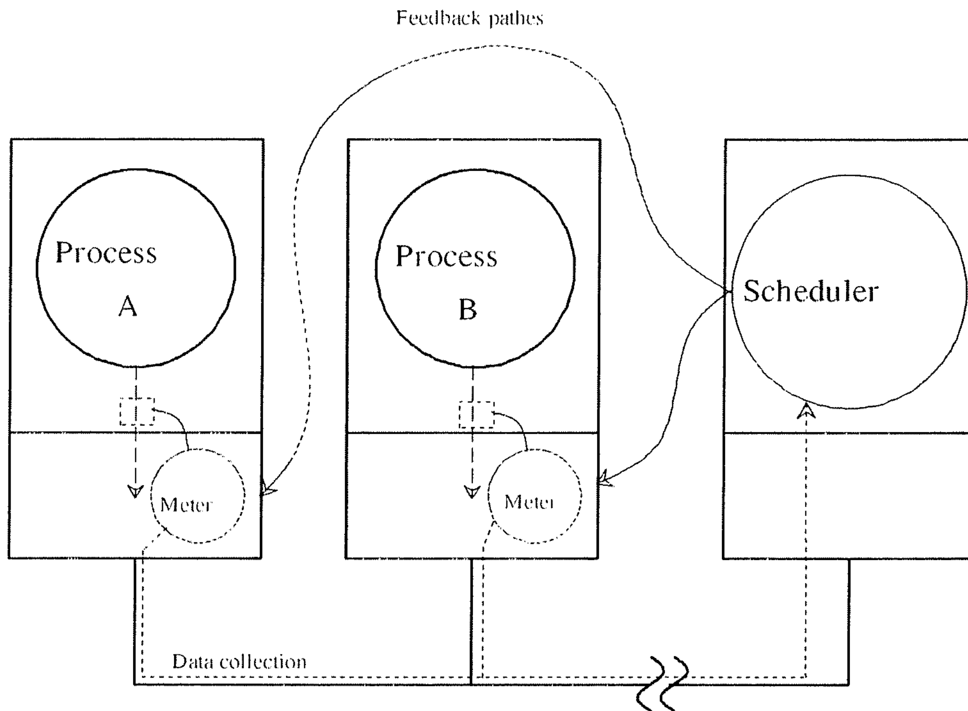

Figure 5.1: Measurement System for Feedback Scheduling

The type of resource scheduling that can make use of the meter data is limited by the frequency with which the data needs to be collected. Activities that occur with about the same frequency as interprocess communications, or with lower frequencies, would be reasonable to measure since the meter traces themselves are messages. Activities more frequent than these would overly degrade system performance. The measurement system can provide information that is gathered from other sources. These other sources could be more traditional performance tools gathering data on machine loading, memory usage, or paging activity. The meter message would be the medium that would carry periodic summaries of these other activities.

6. Conclusion

DPM is a simple tool. Each piece was constructed to provide the needed functions by the most straightforward means. This simplicity provided for its ease of construction (in two operating systems) and for the flexibility of its design.

Transparency to the programmer extended the range of applications. We can measure any program (including existing system services) and programs written in any language. We minimized the restrictions on the use of the tool.

Consistency help in the construction of the measurement systems. It provided a unifying theme throughout the design.

Two design decisions were made in DPM that have inspired some controversy. The first is the lack of user defined event trace types. While this would be easy to add to DPM, we have resisted the temptation so as to maintain transparency. The second decision was the lack of message contents in the meter traces. An argument for the inclusion of message contents comes from monitoring the synchronization protocols in a distributed database system. For example, the last traces we obtained may have come during a commit/abort decision followed immediately by a system crash or deadlock. We would like to be able to know whether a commit or abort was occurring and this might be difficult without knowing the contents of the synchronization. But the inclusion of the PC field in the traces provides us with information on which procedure generated the trace events and this could provide the additional information needed to discriminate between the commit and abort in our example.

Research on DPM continues. The current analyses (communication statistics, paths of causality, parallelism) have provided useful tools for program development. Work is ongoing in the areas of real time monitoring facilities, feedback scheduling, and graphic display techniques.

References

[Baiardi *et al* 83]
F. Baiardi, N. de Francesco, E. Matteoli, S. Stefanini, and G. Vaglini, ''Development of a Debugger for a Concurrent Language,'' *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging*, pp. 98-106 Pacific Grove, Calif., (March 1983).

[Bates & Wileden 83]
P. Bates and J. C. Wileden, ''An Approach to High-Level Debugging of Distributed Systems,'' *Proc. of the ACM SIGSOFT/SIGPLAN Symp. on High-Level Debugging*, pp. 23-32 Asilomar, Calif., (March 1983).

[Joy *et al* 83]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, ''4.2BSD System Manual,'' Computer Systems Research Group Technical Report, University of California, Berkeley (July 1983).

[Lai & Miller 84]
N. Lai and B. P. Miller, ''The Traveling Salesman Problem: The Development of a Distributed Computation,'' Technical Report UCB/CSD 84/212, University of California, Berkeley (October 1984).

[Lauer & Needham 79]
H. C. Lauer and R. M. Needham, ''On the Duality of Operating System Structures,'' *Operating Systems Review* **13**(2) pp. 3-19 (April 1979).

[Macrander 84]
C. M. Macrander, ''Development of a Control Process for the Berkeley UNIX Distributed Programs Monitor,'' M.S. Report, Computer Science Tech. Report UCB/CSD 84/216, Unversity of California, Berkeley (December 1984).

[McDaniel 75]
G. McDaniel, ''METRIC: A Kernel Instrumentation System for Distributed Environments,'' *Proc. of the Sixth Symp. on Operating Sys. Principles*, pp. 93-99 Purdue University, (November 1975).

[Miller, Presotto & Powell 85]
B. P. Miller, D. L. Presotto, and M. L. Powell, ''DEMOS/MP: A Distributed Operating System,'' *submitted for publication*, ().

[Miller 85]
B. P. Miller, ''Parallelism in Distributed Programs: Measurement and Prediction,'' Computer Sciences Technical Report ???, University of Wisconsin-Madison (1985). Submitted for publication.

[Miller *et al* 85]
B. P. Miller, S. Sechrest, and C. Macrander, ''A Distributed Program Monitor for Berkeley Unix,'' *Software - Practice & Experience*, (to appear). Also appears in short form in the 5th Int'l Conf. on Distributed Computing Sytems, Denver (May 1985).

[Powell 77]
M. L. Powell, ''The DEMOS File System,'' *Proc. of the Sixth Symp. on Operating Sys. Principles*, pp. 33-42 Purdue University, (November 1977).

[Schiffenbauer 81]
R. D. Schiffenbauer, ''Interactive Debugging in a distributed computational environment,'' Technical Report MIT/LCS/TR-264, MIT (September 1981).